Obfuscation Techniques for Mobile Agent code confidentiality

Sandhya Armoogum, Asvin Caully
University of Technology, Mauritius
s.armoogum@utm.intnet.mu, asvin.caully@gmail.com

ABSTRACT**:** *Mobile code, typically mobile agent applications, promises many advantages and is suitable to many different types of applications. However, the main hindrance to the advancement of mobile agent technology is security issues such as mobile agent code confidentiality. Existing security mechanisms for mobile agent technology basically allow for protection against code and data integrity changes by using for instance digitally signed mobile agents. However, no viable mechanism for protecting access to the code of the agent from malicious hosts on which the agent arrives. Mobile cryptography which is proposed as the only solution to code confidentiality is too expensive to implement. In this paper, an implementation of code obfuscation has been proposed to provide code confidentiality. Code obfuscation is currently being used in many other fields e.g. software piracy prevention. Thus it is believed that while some more foolproof techniques are devised for code protection against unauthorized access, obfuscation techniques can provide some security. We thus investigate the use of three different obfuscation techniques on java mobile agent code for security.*

Keywords: Mobile Agent, Security, Code Confidentiality, Code Obfuscation, Java.

Received: 1 March 2010, Revised 28 March 2010, Accepted 31 March 2010.

1. Introduction

Mobile code is an important programming paradigm for our increasingly networked world. Unlike mobile computing, in which hardware moves, here mobile code moves to different host machines where the program executes. Already today, the Internet is full of mobile code fragments, such as Java applets, which represent only the simplest form of mobile code. Mobile agents are mobile code that act autonomously on behalf of a user for continuous collecting, filtering, and processing of information. They combine the benefits of the agent paradigm, such as reacting to a changing environment and autonomous operation, with the features of remote code execution. Despite their benefits, massive use of mobile agent is restricted by security issues.

The security concern about mobile agents has been categorised as three distinct threats as follows by [1]: (i) malicious host to agent, (ii) malicious agent to host, and (iii) malicious agent to agent. Hosts executing the agent have complete control on the mobile agents and thus many attacks may be performed by malicious hosts to the mobile agent. The two most important security threats posed by a malicious host is that of code inspection (which is almost impossible to detect – leaves no trace) and code manipulation. Understanding code (code inspection) leads to knowledge about the agent structure and private data such as authentication credentials, private key, electronic cash, and credit card details. Similarly, a host can manipulate the agent code, thus modifying its behaviour or implanting a virus, worm or Trojan horse within the code. The protection of mobile agents from malicious hosts is thus fundamental. Ideally, it is required that the mobile code can execute in an un-trusted environment autonomously i.e. without interactions with its originating site, without the un-trusted host being able to read the agent's code, modify the code or read private data carried by the agent. There are many existing mechanisms for detecting and even protecting agents against tampering. These include the use of detection objects [2], reference states [3,4], state appraisal mechanism [5], server replications [6] and through execution tracing [7,8] to name a few. Currently there is no practical solution to provide mobile agent code confidentiality against malicious hosts. This is a major limitation to agent technology in such applications as ecommerce.

The next sections describe briefly the mobile agent system model of computing followed by the existing schemes for providing mobile code confidentiality. In Section IV, we present the use of code obfuscation for mobile agent code confidentiality. In Section V, implementation details are presented, some results are presented in Section VI and finally in Section VII the conclusion.

2. Mobile Agent System Model and the Malicious Host Problem

Mobile-agents are capable of continued, autonomous operation disconnected from the owner and they migrate to other hosts during their lifetime to perform their task. The use of mobile-agents saves bandwidth and permits off-line and autonomous execution in comparison to usual distributed systems based on message passing as shown in Figure 1 below. Essentially, a mobile-agent consists of code, data and state information needed to carry some computation.

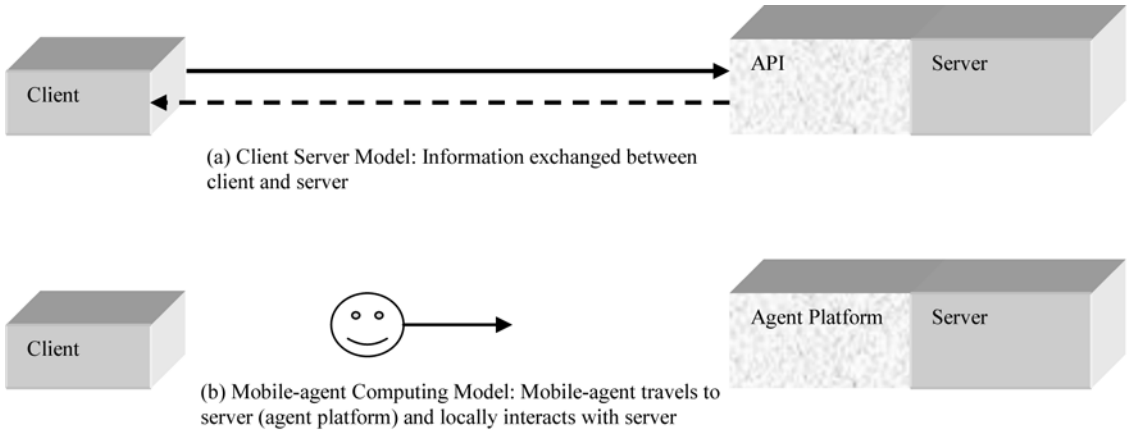

Figure 1. Client-Server model versus Mobile-Agent computing model

Several models exist for describing agent systems [21], [22], [23]. For discussing security related issues though, it suffices to consider a very simple model consisting of the mobile-agent and the agent platform provided by the agent server as described in [1]. The agent platform provides the necessary computational environment for the mobile-agent to operate. The platform from which a mobile-agent originates is referred to as the home platform, and normally is the most trusted environment for a mobile-agent. A simple mobile-agent system model is as depicted in Figure 2.

As can be observed from Figure 1 and 2, mobile agents hop from agent server to agent server and execute locally on the destination agent platform. The agent servers have complete control on the executing mobile-agents and thus many attacks may be performed by malicious servers on the mobile-agent. The malicious server can modify the code, data, and/or state information being carried by the mobile-agent. Likewise the malicious server can inspect the code of the mobile-agent to learn about the decision making strategy of the agent. Again the malicious server may inspect the confidential data such as credit card details or signing key being carried by the mobile-agent. Thus, the protection of mobile-agents from malevolent

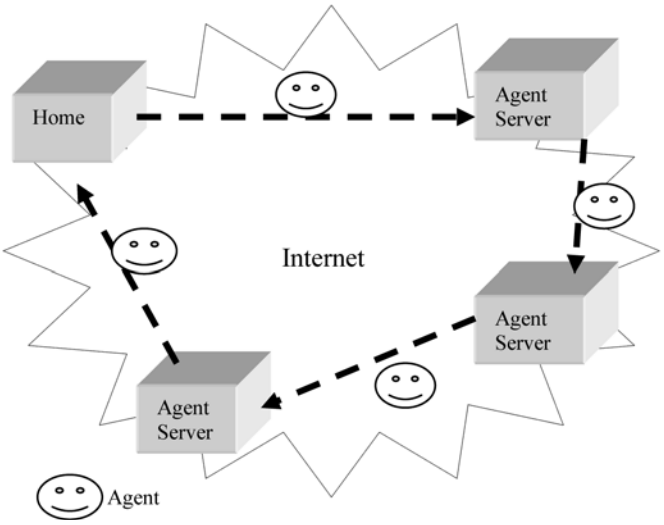

Figure 2. Mobile-agent computing model

agent servers is as important as the protection of the host from malicious mobile-agents. Ideally, it is required that the mobile agent be equipped with security features that enables it to execute in an untrusted environment autonomously (i.e. without interactions with its originating site) and without the untrusted host being able to read and modify the mobile-agent's code and data.

3. Existing Schemes for mobile agent code confidentiality

In e-commerce, a mobile agent usually visits several hosts gathering information about the goods required for purchase. After collecting offers from these seller hosts, the agent is meant to take a decision as to which seller to choose - the agent usually is equipped with a decision making algorithm. After which the agent can use such data as credit card details, e-coins to purchase the goods. If the decision making code is executed on a seller host, the host gets complete access to the code and can completely understand the decision making process of the agent and thus change its offer in order to influence the agents decision such that it is made to choose the malicious seller host for the transaction. Thus, it is important that code confidentiality prevents code inspection.

The simplest mechanism to provide agent code confidentiality would be to allow mobile agent to migrate to known and trusted hosts only. Thus, agents are sent in encrypted form from host to host and are executed only after authentication of the hosts. However, such an approach severely restricts the agent's autonomy, requires a critical mass of infrastructure in order to be used and reduces the number of hosts agent might move to. The following schemes are variations of the same basic idea.

Guan et al in [9] suggest that the network be divided in regions and in each region there be a trusted host called police office (PO). Agents are also divided into two components: master and slave. The slave part of the agent consists of code that can be executed on any host whereas the master part of the agent is critical code such as the decision making strategy which can be executed only on trusted hosts i.e. the PO. Upon entering the region, the agent registers with the PO and sends the slave part of the agent to gather information i.e. on the un-trusted host, the master part remaining on the trusted host. Similarly, following Farmer's perspective [5]: "*An agent's critical decisions should be made on neutral (trusted) hosts*", Silva et al in [10] suggest that the decision making algorithm or any other such critical code and sensitive data be carried encrypted by the agent and is only executed on some specified trusted hosts in the network, assuming that the network consist of un-trusted and trusted hosts. Offers collected from hosts are encrypted using the public key of the agent. Upon reaching the trusted host, the agent sends a message to the home platform such that another agent carrying the private key of the first one is despatched to the trusted host. Offers, sensitive data such as credit card details and decision making code are then decrypted. The code is then executed to allow the agent to choose the best offer. It then migrates to chosen host with unencrypted credit card details to perform the transaction.

It is more practical to have security mechanism that assumes that no host is trusted as it is difficult to know which host to trust. In [11, 12], the main idea is to equip mobile agent systems with additional hardware, which is not under the control of the local system and which can host and execute mobile agents, thus providing a secure execution environment for agents. In this scenario the agents migrate between trusted environments only and access the local resources in a client-server manner. The outside environment cannot interfere with the executing agent except through a restricted interface that is completely controlled by the tamper-resistant module. In [13], it is proposed to use multifunctional chip cards like the Java smart card for realizing cheap and easy trusted hardware instead of using a full blown PC-Card. The main disadvantage of this approach is that it requires that every host be equipped with a secure tamper resistant hardware which is nontrivial as it may be expensive and maintaining such a device can be difficult as well as trust issues regarding the hardware provider. Also, it does not scale up efficiently and given sufficient time and resources, an attacker could violate the protection of any device, and if the trusted hardware's private key is compromised, the attacker gains complete control over the agent sent to the trusted hardware. Moreover, the tamper-resistant devices could become a performance bottleneck. Typically a fully software based solutions is more practical for protecting mobile agents against malicious hosts.

Mobile cryptography attempts to provide provably strong protection to individual mobile agents against tampering and spying attacks. In [14, 15] the use of encrypted programs - mobile agent program can be converted into a ciphered-program such that it can execute on the un-trusted host while remaining in the encrypted form - is proposed as the only way to give privacy and integrity to mobile code (and data). However, mobile cryptography is expensive. Basically, this approach is based on homomorphic encryption scheme and it is very hard to find those schemes for building mobile encryption function. Furthermore those functions can support only restricted execution.

Code obfuscation, is a much more practical approach to providing code confidentiality. Code obfuscation, tries to make the agent's program illegible and data hidden; thus difficult to understand and manipulate. F.Hohl in [16] proposes to generate

an executable agent from a given agent specification such that the generated agent cannot be attacked by read or modify attacks i.e. agent is a black-box. The black-box agent only allows input and output interaction with host i.e. no internal data can be read or modified, thus hosts cannot interfere with the execution of the agent. Currently, there is no known algorithm to fully provide black-box protection. Moreover in [16], a time limited black-box protection is described such that the agent is a black-box only during a limited time interval using code obfuscation methods. After this time, the privacy or a proper execution of the agent cannot be assured. It is true that the host can still read every line of code and the content of every variable but the meaning of these elements to the overall semantics of the application is not easily visible.

4. Code obfuscation for confidentiality

Java programs, delivered as byte code, retain practically all the information of the original source code. This makes them much easier to reverse engineer than traditional applications which are distributed as native code. The Java programming language is open to reverse-engineering because of its well-defined, open, and portable binary format. Reverse engineering can be used by malicious users to understand the mobile agent code or software and determine the decision logic of the mobile agent. Similarly certain security restrictions may be bypassed by malicious host to extract proprietary algorithms and data structures because the mobile agent is fully accessible to the host. It is essential to protect a mobile code against reverse engineering and thus Code Obfuscation is one of the ways to achieve this objective.

Obfuscation is a protection technique for making code unintelligible to automated program comprehension and analysis tools. It works by performing semantic preserving transformations such that the difficulty of automatically extracting computational logic out of the code is increased. The first formal definition of obfuscation was given by [19] where an obfuscator was defined in terms of a compiler that takes a program as input and produces an obfuscated program as output. Two important conditions that need to be preserved while making this transformation are (a) *functionality:* the obfuscated program should have the same functionality (input/output behavior) as the input program, and (b) *unintelligibility:* the obfuscated program should be unintelligible to the adversary in some sense. Some obfuscation methods include data obfuscation (variable, array is obfuscated), control obfuscation (control flow rendered unintelligible). An example of data obfuscation is variable or array splitting. An array could be split into multiple sub-arrays to confuse the reader about its structure. Arrays could also be merged or folded (increase the number of dimensions) or flatten (reduce the number of dimensions). Similarly a Java class can be split into several classes [18].

Our proposed obfuscation algorithm uses the following 3 techniques for obfuscating Java programs: (i) identifier renaming, (ii) control statement insertion or dead code insertion and (iii) comment removal. Most agent development environments are java based, thus the focus of this work is on java code obfuscation.

The existence of discrete names for program constructs such as classes and methods carries with it a certain vulnerability. Languages compiled to machine code have these identifiers stripped out since they are not essential or even required for the execution of the program. Java maintains the naming scheme even in bytecode in order to be able to report meaningful exceptions for debugging purposes, such as where the exception occurred. Replacing class, field, and method names in source code has the potential to remove a great deal of information [20]. In fact, it is probably one of the most effective obfuscations. Method names, for example, very often carry clear and concise explanations of their function (e.g., getName, getSocketConnection, etc.). Our renaming transformation changes all class, field, and method names that is supplied by the operator at runtime. Those identifiers that are renamed, are done so with randomly generated sequences. The second technique inserts useless control statements such as if…else, switch, for and while loop and dead code such as confusing arithmetic expression having nothing to do with the program. This is achieved by a separate class which generates these random statements and acts as a library. These statements are returned through string arrays and written to the Java program at specific positions.Finally the last part of the obfuscation process is carried out by another class which reads through the file and writes to another files except when characters used for commenting are detected i.e. comment removal.

5. Agent Obfuscator Design

The Java Obfuscator consists of 6 modules or classes with each module having very specific functionalities. These modules are: (1) Obfuscator, (2) AcceptInput, (3) GenerateRandomString, (4) SyntaxGenerator, (5) SyntaxChooser, (6) CommentRemover. The Obfuscator module contains the main class of the program. It accepts the input parameter i.e. the java agent file name which needs to be obfuscated. It also performs identifier renaming and insertion of control statements. The flowchart depicting the operation of the obfuscator module is shown in Figure 3 (a) and (b) below.

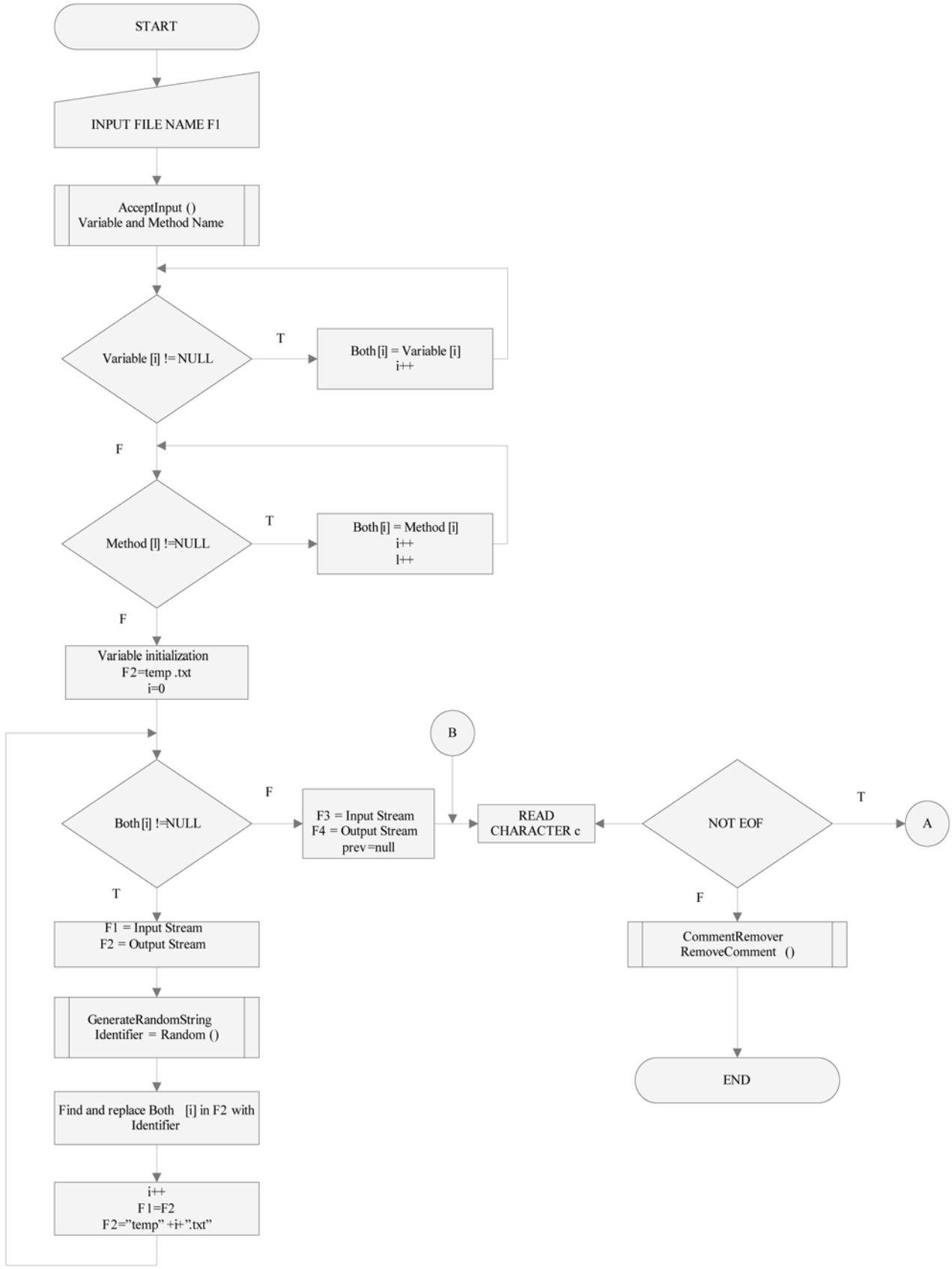

Figure 3(a). Obfuscator module

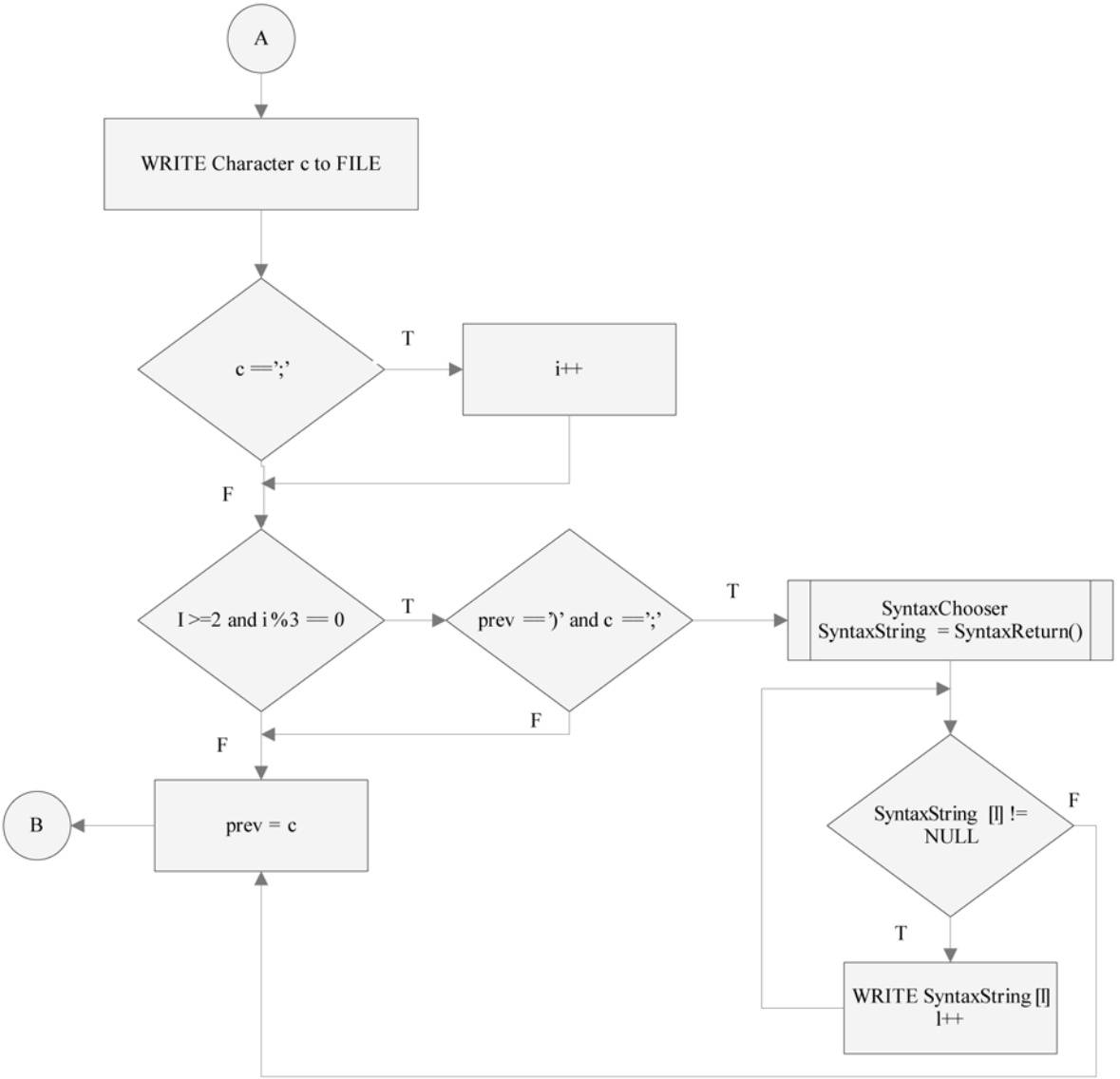

Figure 3(b). Obfuscator module

The AcceptInput module is used to accept the name of the identifiers and methods which needs to be replaced from the operator during runtime of the obfuscator. The AcceptVar() method accepts the variable names and the AcceptMethod() method accepts function names from input stream that are to be renamed. In Figure 4 below is the flowchart for the two functions.

The GenerateRandomString class, as its name suggests, is used to generate random strings of specific length (in our case length is 10). The random string generated by this class is returned to the main program and is used for renaming of variable and function names. Identifier renaming obfuscation is thus achieved with the help of the module. The key idea in this module is to generate a random string from a pre-defined set of characters using the random function in Java. An integer is generated in the range of the length of the predefined set of characters (let us say SET) and this random integer is used to select the character in SET. This character is then stored in a string variable (let us say identifier). This process looped by a specific number of times to finally generate the random string named identifier which is returned to the main class. Figure 5 depicts the GenerateRandomString.

The function of the SyntaxGenerator module is to generate control flow statements or arithmetic expressions. In other words, this means that it acts as a library containing different statements that can be used to be inserted in the Java program to be obfuscated. These statements when inserted into the source Java program, increase the complexity of the program and also

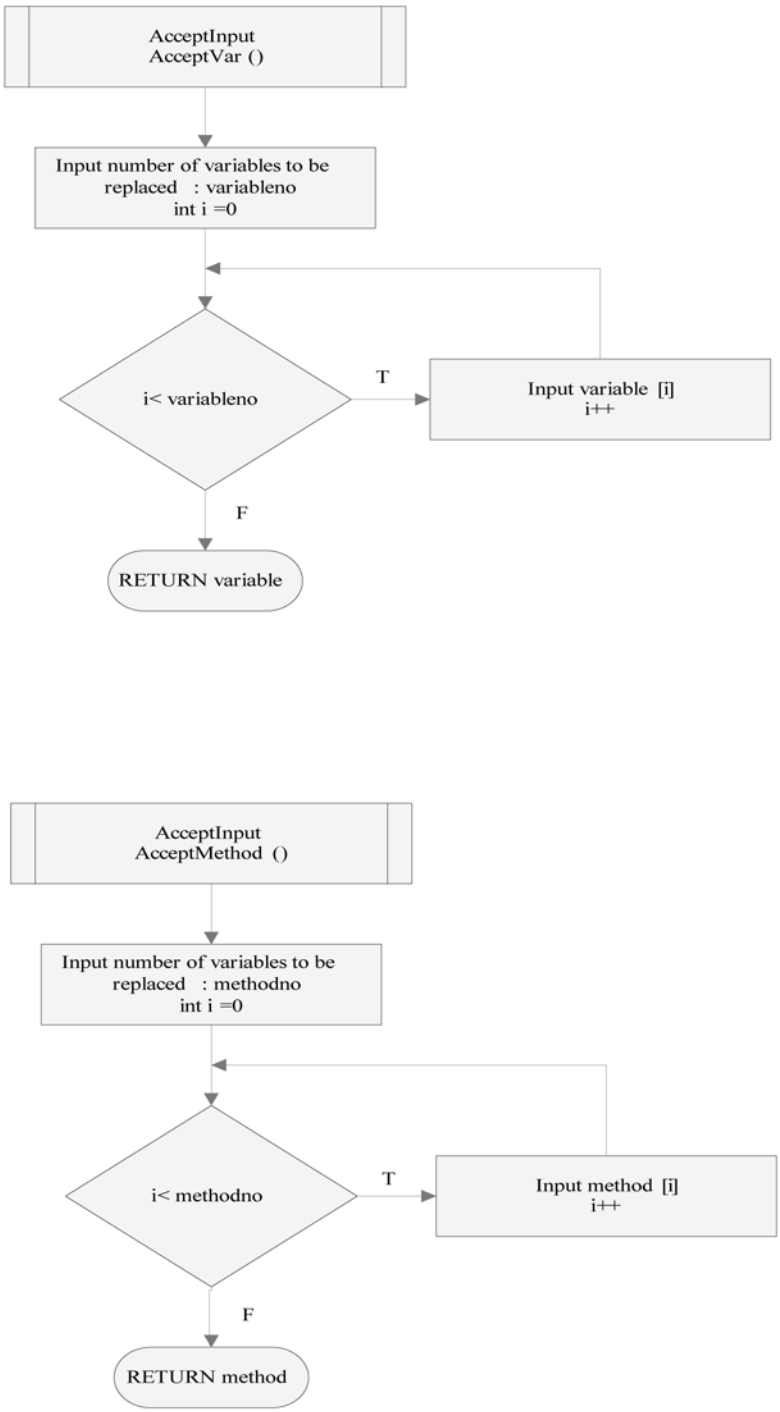

Figure 4. AcceptInput module

makes it harder to reverse-engineer the program to find the logic used. The working principle of the module is based on several functions. Each method generates different control statements. For example, there is the ifcommandGenerator() function which generates a simple if…else statement. The statements generated are stored in predefined string arrays so as to meet the syntax requirements of the Java programming language. The operands to be used in the control flow statements are generated by the GenerateRandomString module. Since the operands must absolutely be defined in the program to be obfuscated, the operands are inserted with their respective data types at the beginning of the array storing the control statement so that when this array is written to the source Java program file, these appear to be defined variables in the source program and do not return "Variable not defined error" during compilation of the obfuscated program.

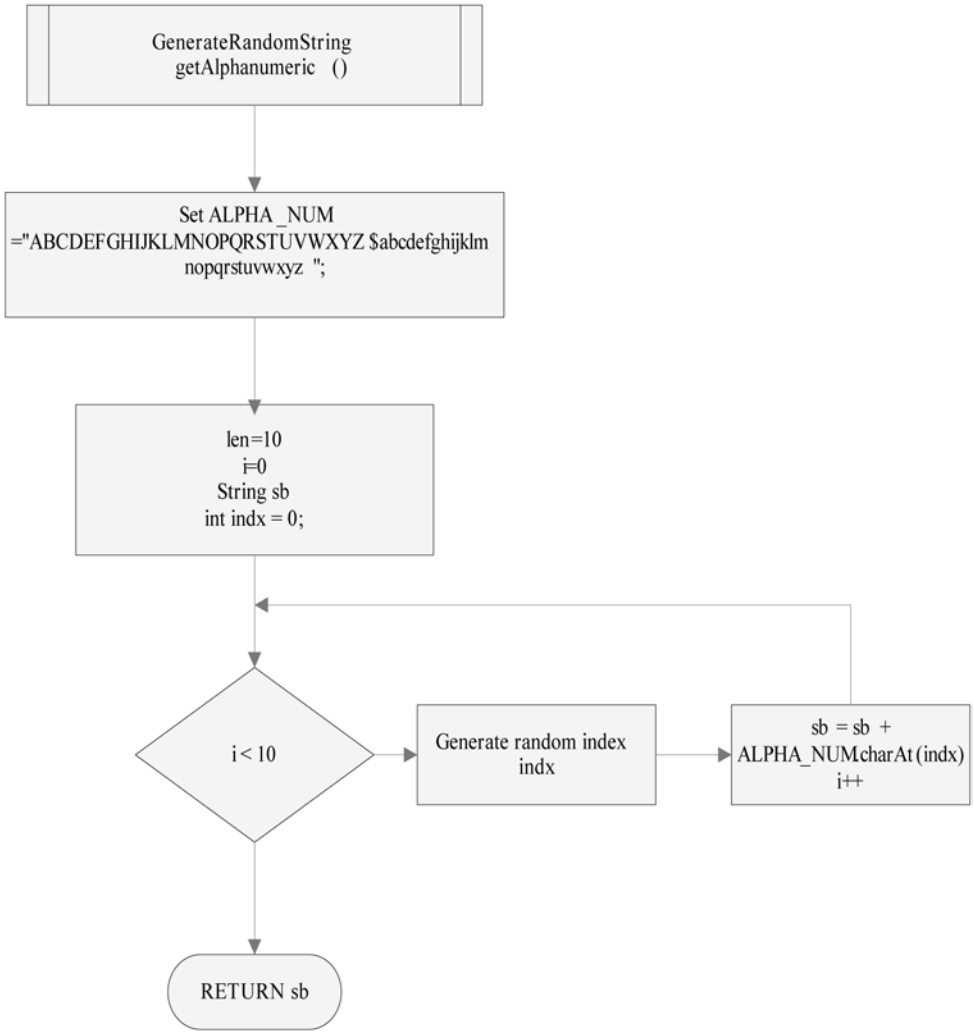

Figure 5. Generate Random String module

The SyntaxChooser module is used to provide different control statement (or dead code) to be inserted in the Java program to be obfuscated. It contains a switch statement which based on a random number generated by the inbuilt random() function of Java, calls the different functions of the SyntaxGenerator module. Figure 6 depicts the Syntax Chooser module.

The CommentRemover is the final step in the Java Obfuscator. Its function is to remove all comments from the Java source program. Many programmers insert comments to explain function of block of codes and this makes it easier for attackers to reverse engineer the program logic. Thus this module removes all comments to help obfuscate the code.

7. Results and analysis

The way the agent code obfuscator was designed resulted in the obfuscated agent code being different every time. Also the output agent code appeared scrambled. However, the obfuscation process was evaluated based on the following metrics: Lines of Code (LOC), Complexity, and Reverse Engineering using Cavaj.

The simplest way to measure the size of a program is to count the lines. This is the oldest and most widely used size metric. The larger the number of lines of code, the more it becomes difficult and time consuming to reverse-engineer a program. But adding more codes might also implies longer transmission times on network. Thus a compromise may have to be made to achieve agent code confidentiality.

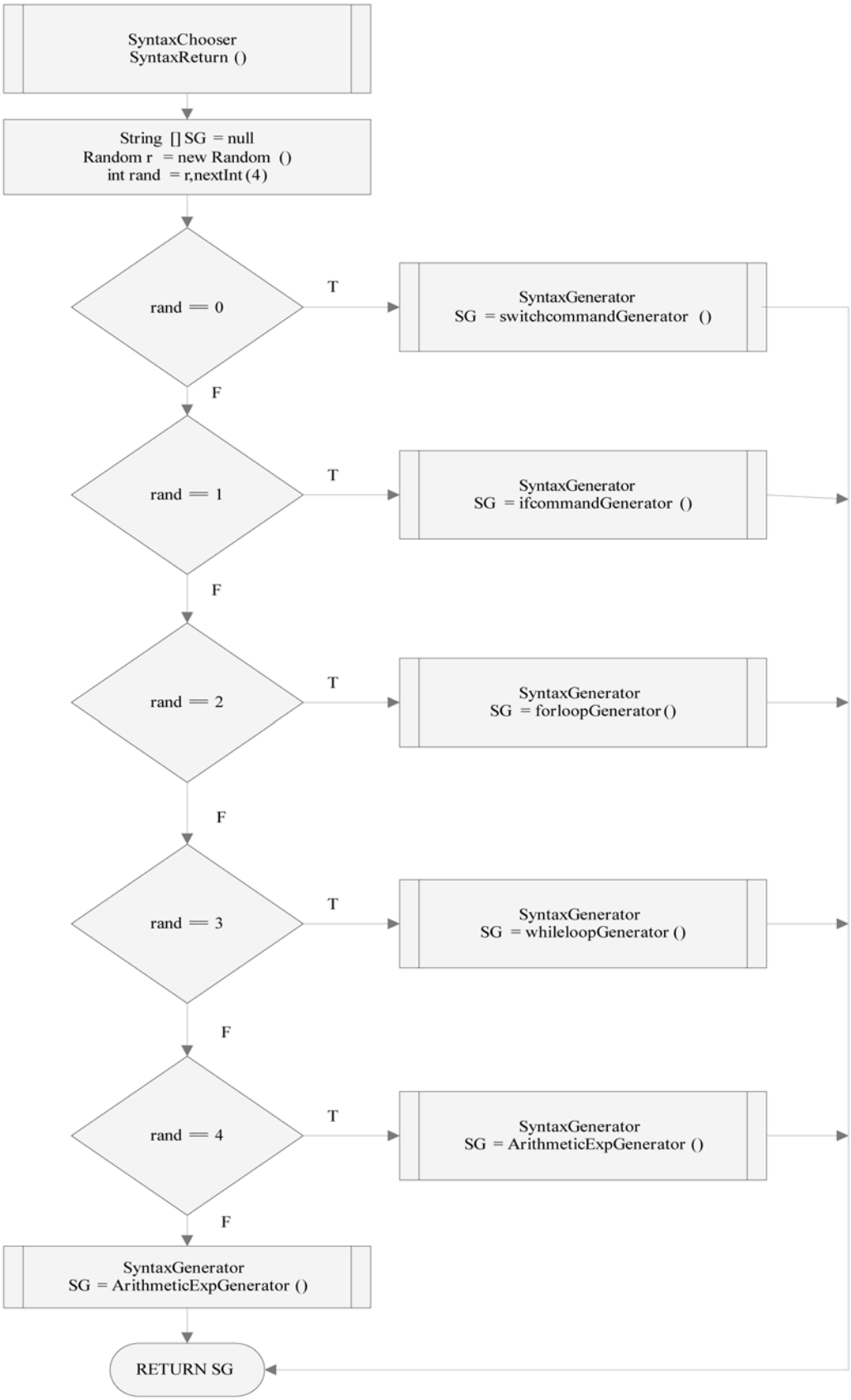

Figure 6. Syntax Chooser module

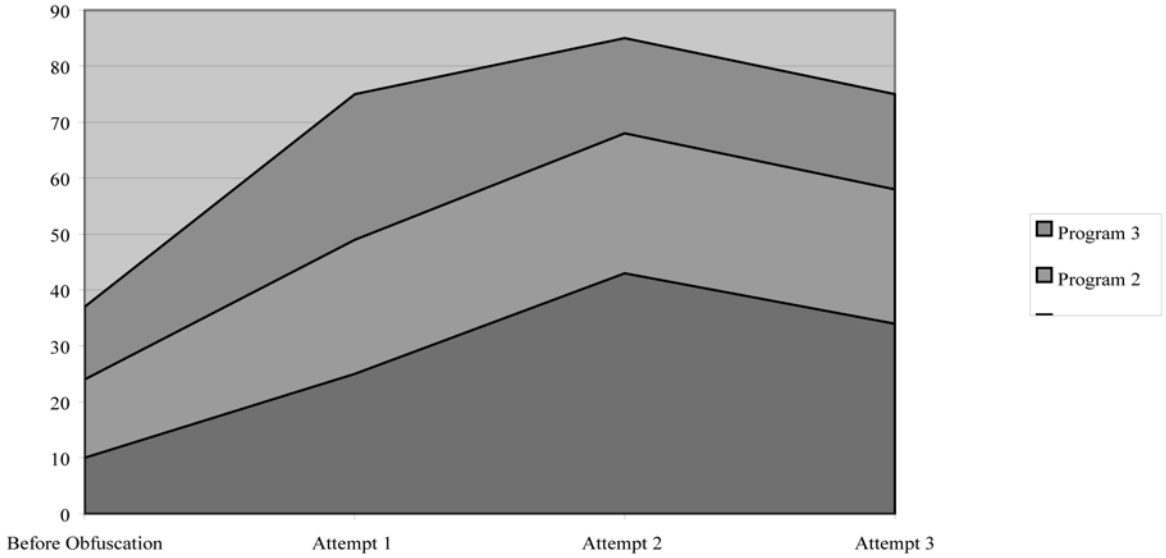

Figure 7. Statement count

There are several ways to count the lines e.g. physical lines, logical lines of code, statements. We have made use of statement count as our metric to measure the impact of obfuscation on the size of the program. It was observed that the statement count of the program increases randomly after obfuscation because obfuscation of the same program generated different outputs. The graph shown in Figure 7 below depicts the statement count for three different programs obfuscated three times independently.

Furthermore, high complexity may result in bad understandability and more errors while reverse engineering a piece of code. The Java Obfuscator does that by inserting dead codes and removing comments to confuse the attacker. In order to compare the complexity before and after the obfuscation, we used the Cyclomatic Complexity. Cyclomatic Complexity (CC), also known as V (G) or the graph theoretic number, is calculated by simply counting the number of decision statements. It is a measure of the structural complexity of a piece of code. A multi-way decision, e.g. the Select Case statement, is counted as several decisions. The basic CC metric however, does not count Boolean operators such as

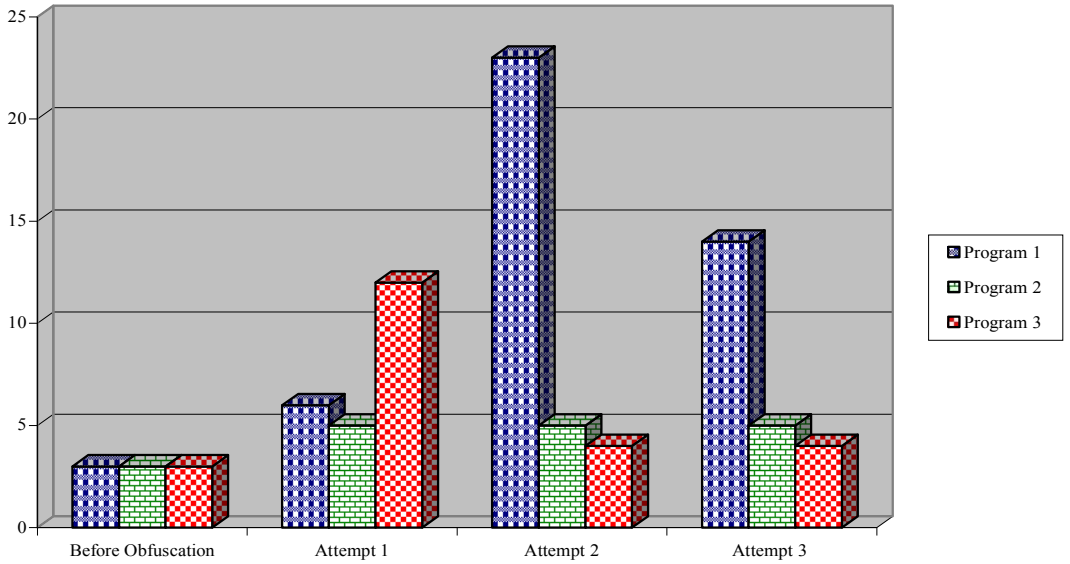

Figure 8. Cyclomatic Complexity analysis

AND and OR, even if they add internal complexity to the decision statements. Cyclomatic Complexity comes in a few variations as to what exactly counts as a decision. Thus CC2 - Cyclomatic Complexity with Booleans ("extended Cyclomatic Complexity") - extends Cyclomatic Complexity by including Boolean operators in the decision count. Whenever a Boolean operator (And, Or, Xor, Eqv, AndAlso, OrElse) is found within a conditional statement, CC2 increases by one. The conditionals considered are: If, ElseIf, Switch, Case, Do, Loop, While, When. For our testing purpose, CC was used. A high Cyclomatic Complexity denotes a complex procedure that's hard to understand, test and maintain. It was observed that Cyclomatic Complexity of the program increases after obfuscation due to the insertion of decision points i.e. control statements in the source program. The graph shown in Figure 8 depicts the recorded CC count for three different programs.

Finally, decompilers are programs that analyze compiled code, and from this, reconstruct the original source code. Decompilation and reverse engineering is often prohibited by software license agreements - but this may not always stop an unscrupulous attacker from analyzing an agent code. Decompilers are freely available for a variety of languages and platforms, including Java! In this work, we analyze the output produced by using a free Java decompiler, namely the Cavaj decompiler, on the obfuscated programs. Cavaj Java Decompiler is a graphical freeware utility that reconstructs Java source code from CLASS files. Several programs were obfuscated and then both the obfuscated and the un-obfuscated programs were decompiled using Cavaj. It is observed that the logic of the unobfuscated code can be easily reversed engineered and understood as it is same as the original code except the identifier names used. However the decompiled version of the obfuscated code is more complex to understand and the program logic remains well hidden. In addition, the switch statement inserted in the obfuscated code could not be properly decompiled. This causes the decompiled code to be confusing and requires the attacker extra effort to be able to reverse-engineer the simple programs.

8. Conclusion

Code confidentiality is an important security requirement for software agent applications involving mobility i.e. where agent has to migrate to other servers (assumed un-trusted) on the network to be able to achieve its goal; typically in e-commerce application where agent may be equipped with a decision making logic and/or payment information. Thus code obfuscation may provide code confidentiality for some time, which in most cases may be enough for the agent to compute on server and move on. We have implemented an agent code obfuscator using three different obfuscation techniques combined such as to provide code confidentiality. Based on our analysis, the obfuscation done indeed was able to increase the complexity of the program rendering it unreadable and difficult to reverse-engineer.

References

[1] Jansen, Wayne., Karygiannis, Tom. NIST Special Publication 800-19 – Mobile Agent Security.

[2] Meadows, Catherine (1997). Detecting attacks on mobile agents. *In*: *Proceedings of the DARPA workshop on foundations for secure mobile code, Monterey CA, USA*, March.

[3] Hohl, Fritz. A Framework to Protect Mobile Agents by Using Reference States.

[4] Fritz Hohl, (1999). A Protocol to Detect Malicious Hosts Attacks by Using Reference States". Technical Report Nr. 09/99, Faculty of Informatics, University of Stuttgart, Germany.

[5] Farmer, W., Guttman, J., Swarup, V. (1996). Security for Mobile Agents: Authentication and State Appraisal," presented at Computer Security ESORICS'96.

[6] Minsky, Y., van Renesse, R., Schneider, F. (1996). Stoller, S. Cryptographic support for fault-tolerant distributed computing. *In*: Proceedings of the Seventh ACM SIGOPS European Workshop, p. 109-114.

[7] Vigna, Giovanni. (1997). Protecting mobile agents through tracing. *In*: *Proceedings of the Third ECOOP Workshop on Mobile Object Systems, Jyv¨askyl¨a Finnland*.

[8] Vigna, G. (1998). Cryptographic traces for mobile agents. *In*: *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag.

[9] Guan, Xudong., Yang, Yiling., You, Jinyuan (2000). POM – A mobile agent security model against malicious hosts", in the proceedings of the fourth international conference on high performance computing in asia-pacific region.

[10] Marques, Paulo., Jorge, Silva., Luis, Moura Silva., Joao, Gabriel. Security mechanism for using mobile agents in electronic commerce.

[11] Wilhelm, U.G. Staamann, S.M., Buttyan, L, (2000). A pessimistic approach to trust in mobile agent platforms, Internet Computing, IEEE 4 (5) 40 - 48

[12] Bennet, S., Yee, A (1997). Sanctuary for mobile agents. *In*: Proceedings of the *DARPA workshop on foundations for secure mobile code*, Monterey CA, USA.
[13] Funfrocken, Stefan. ProtectingMobileWeb-Commerce Agents with Smartcards.
[14] Sander, T., Tschudin, C. (1998). Toward Mobile Cryptography, *IEEE Symp. Security and Privacy*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1998. p. 215-224.
[15] Sander T., Tschudin C., Protecting Mobile Agents Against Malicious Hosts, *In*: Giovanni Vigna (Ed.), Mobile Agent Security, LNCS 1419, Springer Verlag, p. 44-60.
[16] Hohl, Fritz: (1998). Time Limited Blackbox Security: Protecting Mobile Agents From Malicious Hosts, in: Giovanni Vigna (Ed.): Mobile Agents and Security, p. 92-113. Springer-Verlag.
[17] Cavaj, the Java Decompiler. Available: http://cavaj-java-decompiler.en.softonic.com/download
[18] Sonali Gupta 2005, *Code Obfuscation*, Palisade Issue #12, PLYNT Publication. Available : http://palisade.plynt.com/issues/2005Aug/code-obfuscation/ [Accessed 15 November 2009]
[19] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai and K. Yang, \On the (Im)possibility of Obfuscating Programs", In Proceedings of the 21st Annual international Cryptology Conference on Advances in Cryptology, 2001.
[20] Douglas Low(1998), *Protecting Java Code Via Code Obfuscation,* ACM Crossroads, Spring 1998 Issue, Department of Computer Science, University of Auckland.
[21] Fuggetta A., G.P. Picco, and G. Vigna. 1998. Understanding Code Mobility. *IEEE Transactions on Software Engineering, 24(5)*.
[22] FIPA Specification, part 1, version 2.0, Agent Management. Foundation for Intelligent Physical Agents, October 1998
[23] Object Management Group (OMG) Technical Committee (TC). 1997. Mobile Agent System Interoperability Facilities Specification. Document orbos/97-10-05.